Lecture 11

Coordinate systems and Frames

Representing abstract geometric objects

So far, we considered vectors and points as abstract objects(what they are, what operations could be done on them, the meaning on these operations and so on. In a computer graphics program, we need specify specific objects (not abstract objects) to model a scene. To specify a geometric objects, we must consider how it is represented(what we write to specify a given vector for the graphics system).

Starting with vectors: in n-dimensional space, a given vector can be completely specified in terms of any n linearly independent vectors in that space. Since we deal with modeling geometric objects in real world, a three-dimensional space is sufficient for us. Hence, we are interested in how to represent a vector in a three-dimensional space?

Representing vectors in 3-dimensional space

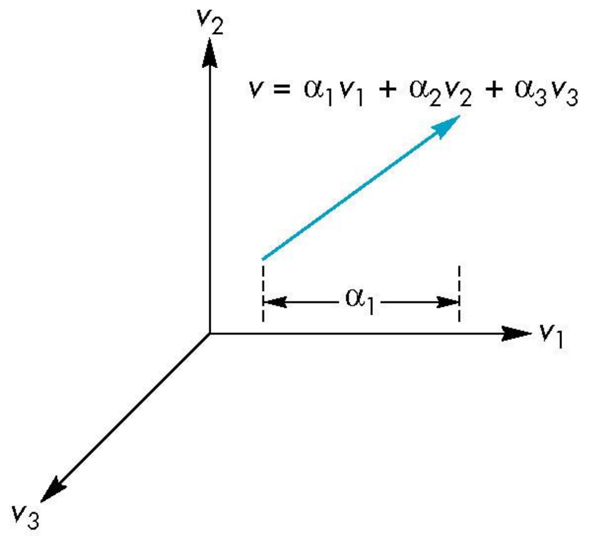
Any vector **w** can be represented in 3-dimenstional space using any three linearly-independent base (usually with unit magnitude) vectors **v**$_1$, **v**$_2$, and **v**$_3$.

$$\boldsymbol{w} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3$$

$where\ the\ scalars\ \ \alpha_1$, α_1, and α_1 are the components of w with respect to the three basis

$$\boldsymbol{a} = \begin{bmatrix}\alpha_1\\ \alpha_2\\ \alpha_3\end{bmatrix} \qquad \boldsymbol{w} = \boldsymbol{a}^T\begin{bmatrix}\alpha_1\\ \alpha_2\\ \alpha_3\end{bmatrix} = \boldsymbol{a}^T\boldsymbol{v},\ \ where\ \boldsymbol{v} = \begin{bmatrix}\boldsymbol{v}_1\\ \boldsymbol{v}_2\\ \boldsymbol{v}_3\end{bmatrix}$$

Now, In our real world problem what three basis we choose to use? Any three linearly independent could be used. The common choice is three perpendicular ones. That is OK for vectors



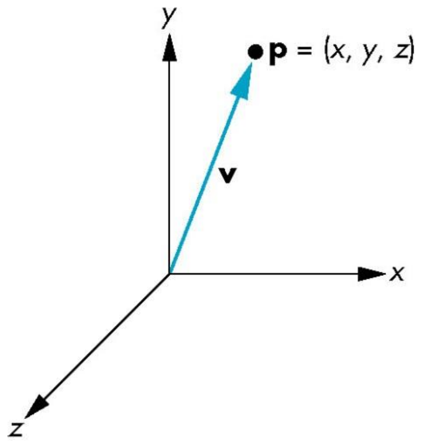
Representing a point

The point is different from vector in that it has a position. Hence, our three basis vectors system is insufficient to represent points and we need a reference point to use according to affine space. If w define a reference point (usually called Origin) in addition to the three bases vectors then we have a Frame.

Having a reference point P_0 and three basis vectors we can now represent vectors and points using affine space as follows

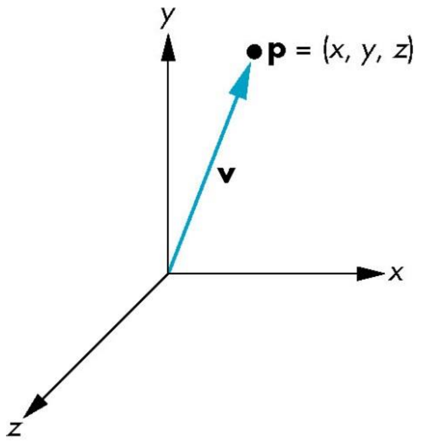
$$\boldsymbol{w} = \alpha_1 \boldsymbol{v}_1 + \alpha_2 \boldsymbol{v}_2 + \alpha_3 \boldsymbol{v}_3 = \boldsymbol{\alpha}^T \boldsymbol{v}$$

$$P = P_0 + \beta_1 \boldsymbol{v}_1 + \beta_2 \boldsymbol{v}_2 + \beta_3 \boldsymbol{v}_3 = P_0 + \boldsymbol{\beta}^T \boldsymbol{v}$$



That is Ok for points and vectors with one problems: We can not write point in matrix notation like vectors. Representing both vectors and points in the same way using matrices is very efficient for a graphics system because it manipulates the two in the same way.

Representing a Point and a Vector in the same way: homogenous coordinates



To represent points and vectors the same way, we use a four dimensional representation for both points and vectors in the three- dimensional space.

$$\alpha_1 = x, \alpha_2 = y, \alpha_3 = z$$

$$v_1 = (1,0,0)^T, v_1 = (0,1,0)^T, v_1 = (0,0,1)^T$$

$$\boldsymbol{v} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3 = \boldsymbol{\alpha}^T\boldsymbol{v}$$

$$\boldsymbol{P} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3 + P_0 = \boldsymbol{\alpha}^T\boldsymbol{v}$$

$$\boldsymbol{v} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3 + 0.P_0 = \boldsymbol{\alpha}^T\boldsymbol{v}$$

$$\boldsymbol{P} = \alpha_1\boldsymbol{v}_1 + \alpha_2\boldsymbol{v}_2 + \alpha_3\boldsymbol{v}_3 + 1.P_0 = \boldsymbol{\alpha}^T\boldsymbol{v}$$

$$v = [\alpha_1 \quad \alpha_2 \quad \alpha_3 \quad 0]\begin{bmatrix}\boldsymbol{v}_1\\ \boldsymbol{v}_2\\ \boldsymbol{v}_3\\ P_0\end{bmatrix}$$

$$P = [\alpha_1 \quad \alpha_2 \quad \alpha_3 \quad 1]\begin{bmatrix}\boldsymbol{v}_1\\ \boldsymbol{v}_2\\ \boldsymbol{v}_3\\ P_0\end{bmatrix}$$

Where $0.P_0$ =0
And $1.P_0 = P_0$.
And this applies for all points

Strictly speaking, the above products are not inner or dot products because the elements of the column vector are not the same type but using homogeneous coordinates allow us using ordinary matrix algebra to represent many operation in software and in hardware

Summary: Frames and Homogeneous coordinates

A frame is defined by an origin (reference) point and a three basis (unit vectors)
A point(position) in the frame is represented in terms of the three basis vector
A vector is represented in terms of the three basis vectors

If the three basis vectors in a frame (any three independent can go but three orthogonal basis are usually are used) are v_1, v_2, v_3, and the origin is P_0 and a vector w are represented as follows:

$$P = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$$
$$W = \delta_1 v_1 + \delta_2 v_2 + \delta_3 v_3$$

The problem is that there is no difference between the representation of the location and the vector given by

$P_2 = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3, w_2 = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

In searching for a clear representation for vectors and locations, the homogenous representation is used

$P_2 = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + P_0$, $w_2 = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

Objects representation from frame to another

If (v_1, v_2, v_3 , P_0) and (u_1, u_2, u_3, Q_0) are two frames, then we can express the basis vectors and reference point of the second frame in terms of the first as

$$
\begin{aligned}
u_1 &= \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3, \\
u_2 &= \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3, \\
u_3 &= \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3, \\
Q_0 &= \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + P_0.
\end{aligned}
$$

=

$$
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = \mathbf{M} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}
$$

where

$$
\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}.
$$

A vector or point in one frame can then be represented in another just by multiplying by a 4x4 matrix. This is the basics of pipelining in OpenGL
Model frame to camera frame (Model-View matrix)
And Camera to clipping window (projection matrix)

Transformation

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \text{ or } \boldsymbol{u} = M\boldsymbol{v}$$

$$\boldsymbol{a} = \begin{bmatrix} \propto_1 & \propto_2 & \propto_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \boldsymbol{\alpha}^T \boldsymbol{v} \quad \textbf{where} \quad \boldsymbol{\alpha} = \begin{bmatrix} \propto_1 \\ \propto_2 \\ \propto_3 \end{bmatrix}$$

$$\boldsymbol{b} = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \boldsymbol{\beta}^T \boldsymbol{u} \quad \textbf{where} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} \; is\ the\ representation\ of\ \boldsymbol{a}\, w.r.t\ \boldsymbol{u}$$

$$\boldsymbol{\beta}^T = \left(\boldsymbol{\alpha}^T (M^{-1})\right) and \; = \boldsymbol{\alpha}^T = \boldsymbol{\beta}^T (M)$$

$$\beta = \left(\left((M^T)^{-1}\right)\alpha\right) \; and \; = \alpha = (M^T)\beta$$

Note the row/vector data layout , transformation matrix and the order of matrices in multiplication. OpenGl thinks always in terms vector data layout. This is also true for homogenous 4-D coordinates

Transformation: Given a object representation in a given frame, what is the representation in the same frame if the object is moved, rotated, scaled, sheared, etc.

Affine (line preserving) Transformation

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

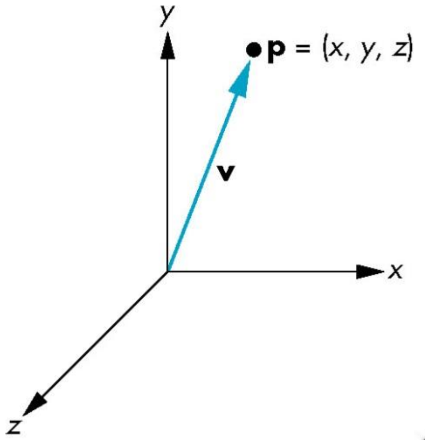
$$\mathbf{T}^{-1}(\alpha_x, \alpha_y, \alpha_z) = \mathbf{T}(-\alpha_x, -\alpha_y, -\alpha_z)$$

Translation

$$\mathbf{S} = \mathbf{S}(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{S}^{-1}(\beta_x, \beta_y, \beta_z) = \mathbf{S}\left(\frac{1}{\beta_x}, \frac{1}{\beta_y}, \frac{1}{\beta_z}\right)$$

Scaling



$$\mathbf{R}_z = \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{R}_x = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{R}_y = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta). \qquad \mathbf{R}^{-1}(\theta) = \mathbf{R}^T(\theta).$$

Rotation

Shear

$$\mathbf{H}_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H}_x^{-1}(\theta) = \mathbf{H}_x(-\theta)$$

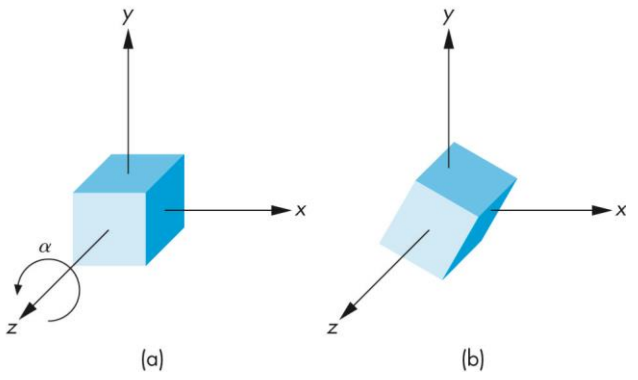
Affine transformation matrices follow the vector data layout in matrix form and multiplication order in OpenGL

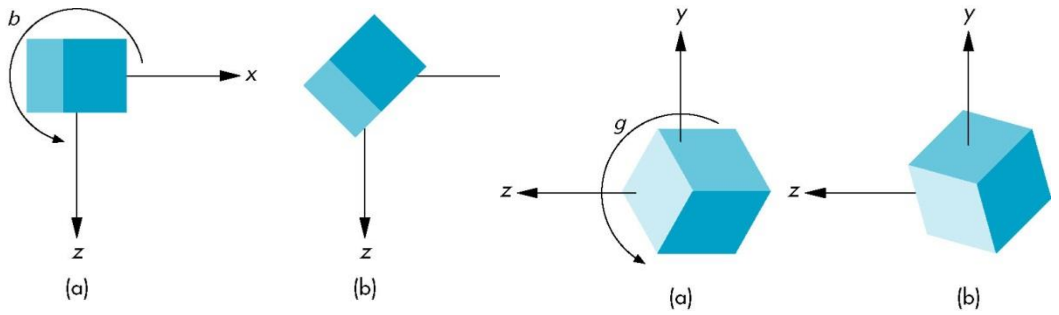
Concatenation of transformation

- If you need to do three transformation
  - First: Scaling (Matrix S)
  - Second: Rotation (Matrix R)
  - Third: Translation (Matrix T)
- You can do that using one matrix which is the multiplication of the three required matrices: M=TRS not M=SRT (assuming we are using the vector data layout, $P_{new}=MP_{old}$, $V_{new}=Mv_{old}$,where P for point and V for vector which OpenGL does)
- Not that the model-view matrix accumulate any transformation until you load the identity

General rotation

An arbitrary rotation about the origin can be composed of three successive rotations about the three axes. The order is not unique, although the resulting rotation matrix is. We form the desired matrix by first doing a rotation about the z-axis, then doing a rotation about the y-axis, and concluding with a rotation about the x-axis.





Total rotation matrix $R=R_xR_yR_z$

A transformation matrix also could be formulated for the rotation about an arbitrary axis

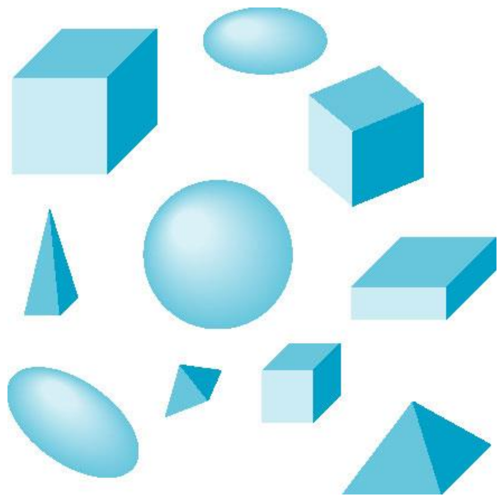
Instance Transformation

Consider modeling a scene composed of many simple objects

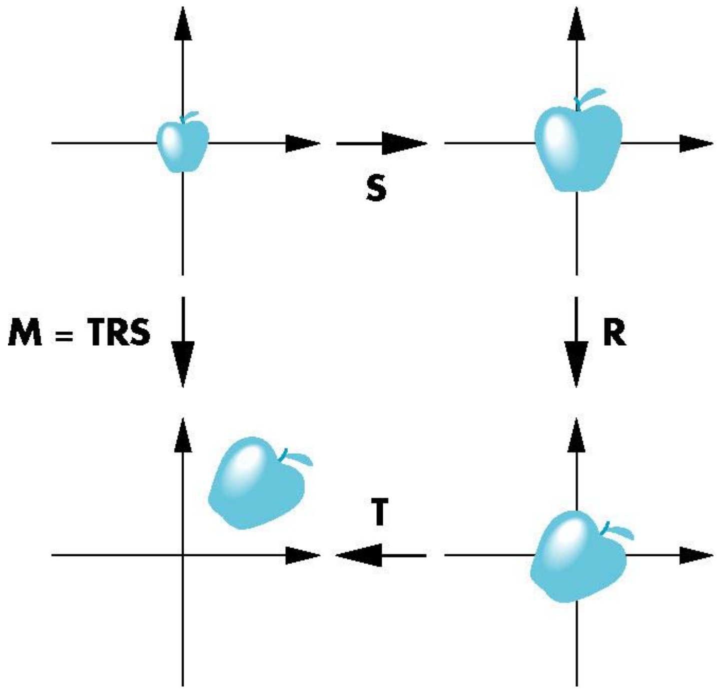
- One option is to define each of these objects, through its vertices, in the desired location with the desired orientation and size.
- An alternative is t o define each of the object types once at a convenient size, in a convenient place, and with a convenient orientation, Each occurrence of an object in the scene is an instance of that object's prototype, and we can obtain the desired size, orientation, and location by applying an affine transformation—the instance transformation—to the prototype.

The instance transformation is applied in the order shown. Objects are usually defined in their own frames, with the origin at the center of mass and the sides aligned with the model frame axes(object coordinates). First, we scale the object to the desired size. Then we orient it with a rotation matrix. Finally, we translate it to the desired location. Hence, the instance transformation is of the form M=TRS

Instance transformation example



A scene composed of instances of simple objects



Order of instance transformation

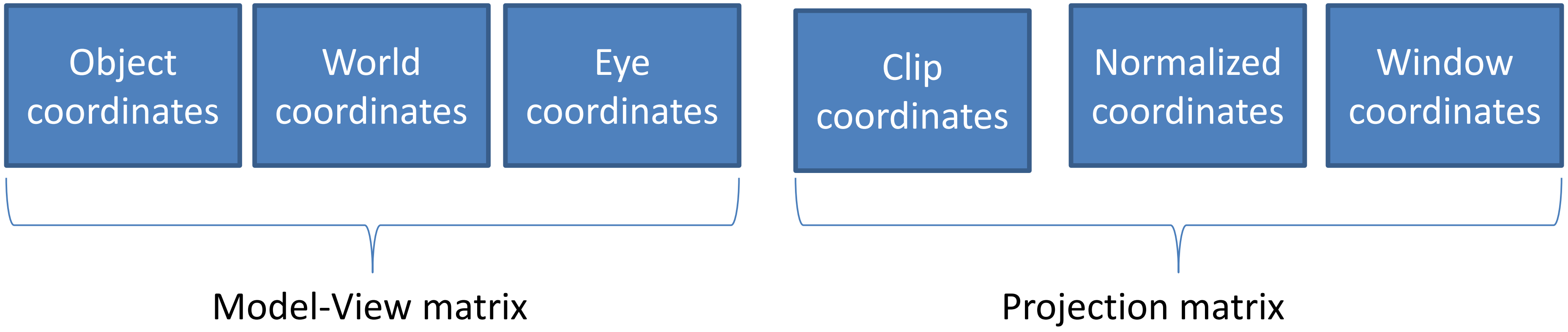
Frames in OpenGL

In a typical OpenGL application, there are six frames embedded in pipelines:

1. Object or model coordinates
2. World coordinates
3. Eye (or camera) coordinates
4. Clip coordinates
5. Normalized device coordinates
6. Window (or screen coordinates)

Typical drawing steps

- Prepare the code the output the object in object coordinates
- Prepare the model view matrix to put the object in its correct position in the world coordinate
- Place the camera correctly in the world coordinate
- Call the code of the object
- Repeat to output the next frame after some changes to reflect effects/motions



Default setting of the OpenGl model-view matrix

The Model-view matrix positions the object frame relative to the eye frame and is a part of the system state which means that there is always a model view matrix. Initially (the default) object frame is the same as the world frame which is the same as the eye frame and the model-view matrix is the 4x4 identity matrix: The three basis vector in the eye-space correspond to(figure a):

- The Up direction of the camera(the Y direction)
- The direction in which the camera is pointing (the negative z direction)
- The third orthogonal direction so that the x,y,z direction form a right-handed coordinate system

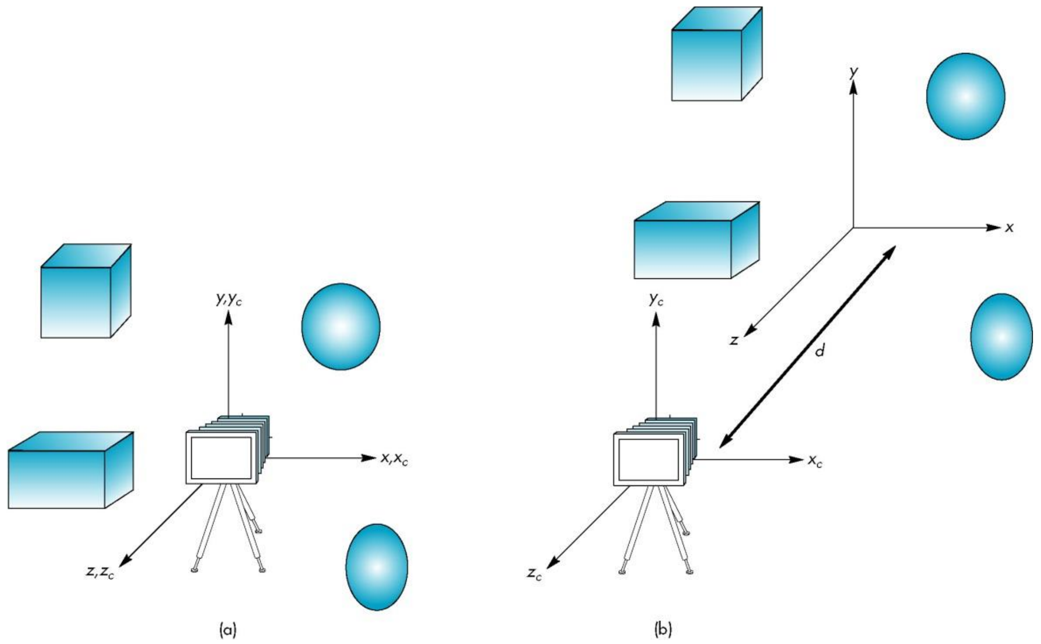


Figure b, the camera is moved along the z axis to see all the objects in the scene by changing the model-view matrix. The same effect can be done by changing the object positions w.r.t the camera through the model view matrix which unnatural compared to the first thinking

OpenGL Transformation matrices

- In OpenGL, several matrices are part of the state. We will use only the model-view matrix here. The matrix state is manipulated by a common set of functions, and we use the function glMatrixMode to select the matrix to which the operations apply.
- The Selected or current matrix(c) receive the operation but the Current Transformation Matrix(CTM) in OpenGL is always the multiplication of the model-view matrix and the projection matrix. You specify a vertex V and the pipeline produces (CTM)V
- In OpenGL, the model-view matrix normally is an affine-transformation matrix and has only 12 degrees of freedom. The projection matrix, is also a 4 x 4 matrix, but it is not affine. OpenGL allows you to work with arbitrary 4 x 4 matrices and make use of all 16 degrees of freedom.

The three transformations supported in most systems including OpenGL are translation, scaling with a fixed point of the origin, and rotation with a fixed point of the origin. Symbolically, we can write these operations in post-multiplication form as $C \leftarrow CT$, where C is the current matrix and T is the transformation (OpenGl allows only the post-multiplication of the current matrix)

Affine transformation in OpenGL

The post-multiplication form
C←CT for translation
C←CS for scaling
C←CR for translation
Where C is the current transformation matrix

```
glRotatef(angle, vx, vy, vz);
glTranslatef(dx, dy, dz);
glScalef(sx, sy, sz);
```

```
glMultMatrixf(pointer_to_matrix).
```

All are post-Multiplication and the pointer is a pointer to a one-dimensional array of 16 entries organized by the columns of the desired matrix.

The load form
C←T for translation
C←S for scaling
C←R for translation
Where C is the current transformation matrix

```
glLoadMatrixf(pointer_to_matrix);
```

```
glLoadIdentity();
```

Transformation Concatenation in OpenGL

The rule in OpenGL is this: The transformation specified last is the one applied first. A little examination shows that this order is a consequence of multiplying the CTM on the right by t he specified affine transformation and thus is both correct and reasonable.

Example: we can perform a rotation about a fixed point , other than the origin, by first moving the fixed point to the origin, then rotating about the origin, and finally moving the fixed point back to its original location.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(4.0, 5.0, 6.0);
glRotatef(45.0, 1.0, 2.0, 3.0);
glTranslatef(-4.0, -5.0, -6.0);
```

$$\mathbf{C} \leftarrow \mathbf{I},$$

$$\mathbf{C} \leftarrow \mathbf{CT}(4.0, 5.0, 6.0),$$

$$\mathbf{C} \leftarrow \mathbf{CR}(45.0, 1.0, 2.0, 3.0),$$

$$\mathbf{C} \leftarrow \mathbf{CT}(-4.0, -5.0, -6.0).$$